ORNL/TM-2009/120

COMPLEX VERSION OF HIGH PERFORMANCE COMPUTING LINPACK BENCHMARK (HPL)

R. F. Barrett, Oak Ridge National Laboratory
T. Chan, Chinese University of Hong Kong
E. F. D'Azevedo, Oak Ridge National Laboratory
E. F. Jaeger, Oak Ridge National Laboratory
K. Wong, University of Tennessee
and R. Wong, Chinese University of Hong Kong

Date Published: April, 2009

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6367
managed by
UT-Battelle, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

List of Figures

List of Tables

ACKNOWLEDGEMENTS

We express our appreciation to the reviewers of the version of this paper that appeared as [6]. Their careful reading and suggestions significantly improved the presentation of this work.

This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725. This research used resources of the Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Summer Internships for T. Chan and R. Wong were supported by the Department of Mathematics, The Chinese University of Hong Kong (CUHK). Internship opportunity was provided by the Joint Institute for Computational Sciences (JICS), the University of Tennessee, and the Oak Ridge National Laboratory.

ABSTRACT

This paper describes our effort to enhance the performance of the AORSA fusion energy simulation program through the use of High Performance LINPACK (HPL) benchmark, commonly used in ranking the top 500 supercomputers. The algorithm used by HPL, enhanced by a set of tuning options, is more effective than that found in the ScaLAPACK library. Retrofitting these algorithms, such as look-ahead processing of pivot elements, into ScaLAPACK is considered a major undertaking. Moreover, HPL is configured as a benchmark, and only for real-valued coefficients. We therefore developed software to convert HPL for use within an application program that generates complex coefficient linear systems. Although HPL is not normally perceived as part of an application, our results show the modified HPL software brings a significant increase in performance of the solver when simulating the highest resolution experiments thus far configured, achieving 87.5 TFLOPS on over 20,000 processors on the Cray XT4.

1 Introduction

The next step toward fusion as a practical energy source is the design and construction of ITER (www.iter.org), a device capable of producing and controlling the high performance plasma required for self-sustaining fusion reactions, i.e. "burning plasma." Computer simulation is providing significant insight into how this can be accomplished [5]. For example, the AORSA [20, 21, 22] (**A**ll **OR**ders **S**pectral **A**lgorithm) simulation program, developed within the Scientific Discovery through Advanced Computing (SciDAC) *Numerical Computation of Wave Plasma-Interactions in Multi-dimensional Systems* project[1] has demonstrated how electromagnetic waves can be used for driving current flow, heating and controlling instabilities in the plasma. AORSA provides high-resolution, two-dimensional solutions for mode conversion and high harmonic fast wave heating in tokamak plasmas.

The computer program takes advantage of new computational techniques for massively parallel computers to solve the integral form of the wave equation in two dimensions without any restriction on wavelength relative to orbit size, and with no limit on the number of cyclotron harmonics retained.

Parallelism in AORSA is centered around the construction and solution of a large $N \times N$ system of complex linear equations of the form $Ax = b$. For problems of interest, the increasingly better physics model and higher resolution of the mode conversion layer generates linear systems exceeding rank $N = 500,000$. Thus it is the solution of these systems, with computational requirements of $O(N^3)$ that presents the greatest computational challenge for managing runtime of an AORSA simulation.

Originally configured for the general solver (subroutine `PZGESV`) provided by ScaLAPACK [10, 8], excellent performance was achieved on on several computing platforms, including an IBM Power3-based computer at NERSC (named Seaborg), and an SGI Altix 3800 (Ram), Cray X1 (Phoenix), and a Cray XT3 (Jaguar) located at the at National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL). However, as the scale of the target simulations increased corresponding to the increased scale of available computing resources, performance began to diminish. In particular, early evaluation of AORSA2D on the Cray XT4 (Jaguar, upgraded from an XT3) using the vendor tuned ScaLAPACK library achieved a lower than expected performance of about 38% of peak performance on over 10,000 cores (illustrated in Figure 1). This expectation is in comparison to the realized performance of the High Performance LINPACK (HPL) benchmark [13, 17], the metric by which the world's fastest computers are ranked (www.top500.org), that achieved 86% of peak on the dual-core configuration of Jaguar, and 79% of peak on the recently upgraded quad-core configuration[2].

[1] Web site for Center for Simulation of Wave-Plasma Interactions (CSWPI) is www.scidac.gov/fusion/CSWPI.html.

[2] This value is expected to increase as the machine matures.

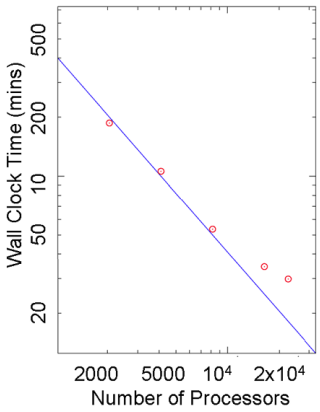


Fig. 1: Solution time for ScaLAPACK for 350×350 ITER simulation.

HPL implements sophisticated algorithms such as look-ahead processing of pivot elements, hybrid left-looking and right-looking factorization algorithms, and methods to reduce communication latency by combining several messages. More details of the similarities and difference between ScaLAPACK and HPL follows. We considered retrofitting similar look-ahead algorithms into ScaLAPACK to be a major undertaking. Our approach instead was to pursure adaptation of the freely available (though with license and copyright) version of the HPL benchmark for use by AORSA. We decided on using a semi-automatic translation process to minmize human errors and to more easily incorporate enhancements in future versions of HPL. In this paper we report on our efforts of the adaptation and the resulting gains in performance.

After a discussion of related work, we begin with a comparison of ScaLAPACK and HPL, and describe how they are used within AORSA. Next, we describe the conversion of HPL to accept (double precision) complex coefficient linear systems. We then compare the performance of ScaLAPACK and HPL, within the context of AORSA simulations, on two distinct computing architectures. Lastly, we offer our conclusions and suggestions for further improvements for the performance for solving the linear system in AORSA.

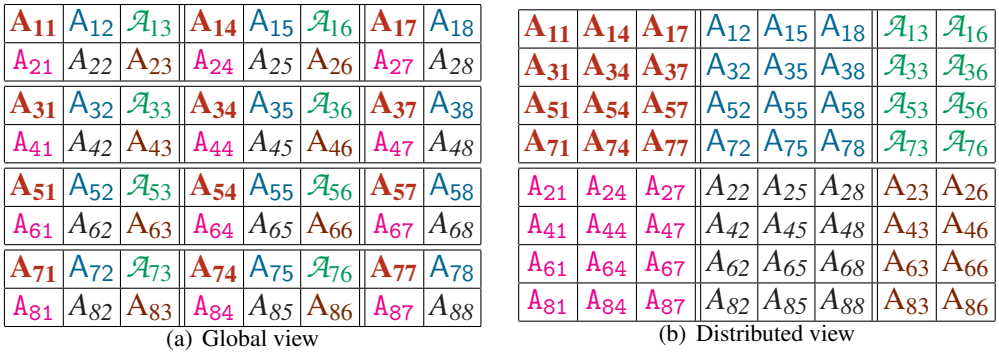
2 Related Work

Solving dense linear systems of equations has been an area of focus of computational science from its beginning. Several computer vendors provide such capabilities within some form of a tuned scientific library, such as IBM's ESSL, Cray's LibSci, Intel's Math Kernel Library (MKL), and SGI's SCSL. The most popular freely available library is LAPACK [3]. (The GNU Scientific Library does not provide Fortran interfaces to their algorithms.) Further, and even more commonly, vendors provide implementations of the BLAS [23, 15, 14], which are typically used as the computational building blocks for linear solvers. They can be tuned to take significant advantage of the computing system capabilities. Highly tuned freely available versions include [19, 29].

Several computer vendors, including those listed above, provide parallel processing capabilities for solving dense linear systems of equations. Our understanding is that these implementations typically are based on ScaLAPACK algorithms. PLAPACK [27] uses the same algorithm as ScaLAPACK, but provides an interface designed to ease the burden of distributing the equations to 2D block cyclic format. (This task is similarly accomplished in AORSA using parallel BLAS (PBLAS) [11] routine `PZGEADD`.)

Cray provides a mixed-precision algorithm [9] in their iterative refinement toolkit, a component of their LibSci library. As it is also based on the ScaLAPACK factorization algorithm, we are in the process of implementing the mixed-precision algorithm using HPL, which also requires the code conversion techniques described in this paper. We intend to describe this work in a subsequent report.

We are not aware of any previous attempts to create a user interface to HPL functionality, nor are we aware of previous work that would have simplified the software conversion of HPL to accept complex coefficients, nor to create user interfaces from a non-externally callable program. We used commonly available tools such as Awk [1] and shell scripts for their simplicity. Other scripting languages, such as Perl [28] or Python [24], could also be used. C++ templates and function overloading might be desirable for creating a more general implementation. We used C99 for portability, expediency, simplicity, and native support of complex numbers, in addition to the fact that HPL is written in standard C.



(a) Global view

(b) Distributed view

Fig. 2: Global and distributed views of two-dimensional block cyclic distribution of matrix across 2×3 logical grid of processors.

3 ScaLAPACK and HPL

Both ScaLAPACK and HPL require a two-dimensional block-cyclic data distribution[3], illustrated in Figure 2. Both are renowned for portability across a broad variety of parallel processor-based architectures, attributable to their use of standard language constructs (ScaLAPACK using Fortran, HPL using C) and MPI [25]. Both solve the linear equation ($Ax = b$) by first factoring the matrix into upper- and lower-triangular factors,

$$Ax = PLUx = b, \tag{1}$$

then compute the solution vector using forward then backward substitution.

HPL is a portable reference implementation of the High Performance LINPACK Benchmark [17]. HPL generates and solves a dense linear system with double precision real-valued coefficients. ScaLAPACK includes a subset of LAPACK functionality [3] adapted for use on parallel architectures: the solution of linear systems, linear least squares problems, and eigenvalue problems, for dense, banded, and triangular matrices. Widely accepted by the scientific computing community, ScaLAPACK (and its component software[4]) is usually provided by vendors on most architectures (with examples discussed above). ScaLAPACK was designed to mimic LAPACK in the use of blocked algorithms [16]. ScaLAPACK does not call MPI directly but uses PBLAS and the Basic Linear Algebra Communication Subroutines (BLACS) to achieve parallelism and for interprocessor communication. While this functional component structure is an effective software development strategy (see Figure 3), it enforces a synchronous

[3] This decomposition ensures a balanced load across the parallel processes throughout execution, and enables the use of level 3 BLAS functionality. The latter allows for strong data re-use, a requirement for achieving strong performance on cache-based processors.

[4] LAPACK, BLAS [23, 15, 14], and BLACS [18].

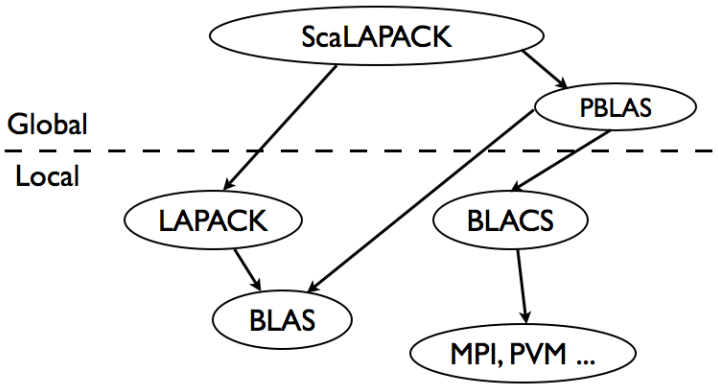


Fig. 3: ScaLAPACK software structure.

configuration for its algorithms. That is, because the algorithms used in ScaLAPACK are patterned after LAPACK block algorithms, each parallel process must participate in the same call to ScaLAPACK or PBLAS. For example, a call to `PZGEMM` (double precision complex matrix multiplication) requires that all processors participate even if they have no data associated with the computation. The subsequent call to PBLAS cannot proceed until all processors have completed the previous call to PBLAS. This bulk synchronous parallel paradigm [7] simplifies code development but introduces unnecessary barriers.

Since HPL was designed to be a benchmark, the HPL driver has several parameters to select different variants of the basic LU factorization algorithm with row pivoting. HPL offers left-looking, right-looking, and Crout recursive variants. HPL uses a block panel algorithm and the panel factorization is on the critical path. Sending the pivot information ahead (also called look ahead) to the next processor before performing updates to the trailing matrix is one way to reduce this delay [26]. HPL reduces communication overhead by combining several operations such as pivot search, row swap, and broadcast operations all integrated into a single communication step. Moreover, to achieve the best performance in communication, HPL implements six variants of broadcast operations: increasing ring, modified increasing ring, increasing two-ring, modified increasing two-ring, bandwidth-reducing, and modified bandwidth reducing. There are many other options that influence the memory alignment, how L and U factors are stored, and different ways of performing row swaps. The details of the various algorithmic options of HPL are documented in [17, 12]. Significant experimentation with these options is constrained by our limited access to Jaguar at the necessary scale. Thus for our purposes, we configured HPL with options that were used in the contribution to the TOP500 list.

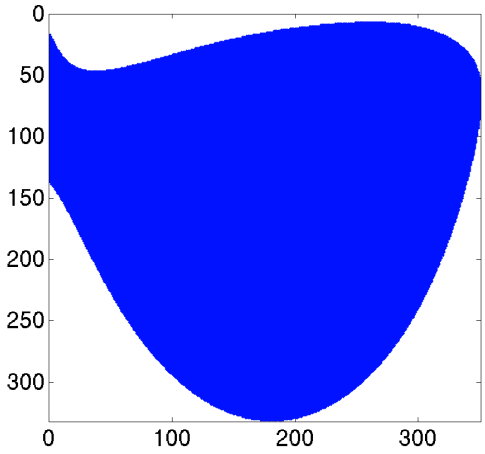


Fig. 4: AORSA computational spaces. The axes represent the grid resolution. The square region is the Fourier space, while the shaded region represents the fusion energy device within that region.

3.1 The linear system in AORSA2D

AORSA models the heating response of plasma due to radio frequency (RF) waves. The plasma state is described by a distribution function $f_s(\mathbf{r},\mathbf{v},t)$. For RF application, the fast wave time scale leads to effective approximation of the electric field, magnetic field and distribution function as a time-averaged equilibrium part $(\mathbf{E}_0,\mathbf{B}_0,f_s^0)$ and a rapidly oscillating time-harmonic part, $(\mathbf{E}(\mathbf{r})\exp(-i\omega t),\mathbf{B}(\mathbf{r})\exp(-i\omega t),f_s^1(\mathbf{r},\mathbf{v})\exp(-i\omega t))$. The time harmonic terms satisfy the generalized Helmholtz equation,

$$-\nabla\times\nabla\times\mathbf{E}+\frac{\omega^2}{c^2}\left(\mathbf{E}+\frac{i}{\omega\varepsilon_0}\mathbf{J}_p\right)=-i\omega\mu_0\mathbf{J}_{antenna} \tag{2}$$

$$\mathbf{J}_p(\mathbf{r},t)=\int_{-\infty}^{t}dt'\sum s\int d\mathbf{r}'\sigma(f_s^0(E),\mathbf{r},\mathbf{r}',t,t')\cdot\mathbf{E}(\mathbf{r}',t') \tag{3}$$

where ω is the frequency of wave, $\mathbf{J}_p$ is the plasma current induced by the wave fields, $\sigma(f_s^0,\mathbf{r},\mathbf{r}',t,t')$ is the plasma conductivity kernel.

Fourier modes are used as basis functions to represent the electric field. Collocation on a $M\times M$ rectangular grid is used to construct a complex linear system of size $N=3\times M\times M$. A reduced linear system can be constructed by transforming the linear system (using Fast Fourier Transform) into the real physical space and consider only collocation points within the plasma region (illustrated in Figure 4). Typically about 20-30% of grid points are in the vacuum region. Thus the memory required still grows as $O(M^4)$ and computation work grows as $O(M^6)$.

Another version, AORSA3D, provides fully three-dimensional (3D) solutions of the integral wave

equation for minority ion cyclotron heating in three dimensional stellarator plasmas. By combining multiple periodic solutions for individual helical field periods, it is possible to obtain complete 3D wave solutions valid over the entire volume of the stellarator for arbitrary antenna geometry. AORSA3D will require even higher computational resources.

4 Conversion of HPL to complex coefficients

The HPL code, developed in 2001, was written using the C programming language. It was designed to solve a randomly generated dense linear system using LU factorization in 64-bit (double precision) arithmetic on distributed-memory computers. We modified the HPL source code to solve a given double precision *complex* linear system to replace the ScaLAPACK routine `PZGESV`. HPL provides options for improving performance based on architecture capabilities. Options include the depth of the selected look-ahead algorithm, the use of hybrid left-looking and right-looking algorithms for panel update, the topology of the logical processor grid, and several variants of coding for efficient message passing in MPI.

There were two main issues in the conversion of HPL to mimic the functionality of complex solver `PZGESV` in ScaLAPACK. The first was the conversion of the C code to use double-precision complex data type and the second was to provide a Fortran callable interface to the HPL solver that remained compatible with ScaLAPACK and PBLAS. (PBLAS functionality is used in AORSA to distribute the equations to the required 2-D block cyclic distribution.)

Interface code was written to smoothly couple HPL with ScaLAPACK. For example, HPL and ScaLAPACK must be initialized with the same MPI communicator and the same context of the logical processor grid. HPL stores the right hand side vector as an extra column and uses an extra row to hold a temporary vector. Moreover, HPL attempts to increase memory performance by aligning data structures to start on a cache line. Therefore, the interface code queries HPL for the amount of storage required before allocating the array in the Fortran code. During the panel factorization process, HPL performs the row pivoting and rearrangement only to the right unfactored part of the matrix for higher efficiency. The resulting *LU* factors are not strictly identical to the factorization obtained by `PZGETRF` in ScaLAPACK. Thus after HPL has completed, the interface code extracts the pivot vector and performs an extra pass over the *LU* factors to rearrange the data to be compatible with ScaLAPACK. For the needs of the fusion application, the righ hand side vector was solved only once and this extra rearrangment was not needed.

The C99 language standard, which defines double-precision complex data types and complex arithmetic expressions, is supported by the Portland Group `pgcc` compiler on the Cray and `gcc` compiler on the Linux cluster. Strict type checking on function prototypes by the `gcc` compiler aided identification of code that required modification. Awk and shell scripts were used to generate a complex version of HPL in a semi-automatic manner. The number of files that were modified is significant: 68 of the 79 files (one function per file in general) in the source tree, 6 of the 17 files in the test programs, and 13 of the

19 header files (*.h) were modified. Those not needing modification were typically involved in matrix indexing requirements or general process management (e.g. HPL_Abort and HPL_grid_info).

In general the changes were minor, mostly changing keyword double to zcomplex, which motivated our use of an automated approach for making the necessary modifications. However, it was important to manually examine each function in order to determine that complexity was overlooked. Direct intervention on a small number of routines was still required.

4.1 Details

In this section we present the details of the conversion and list the special cases not handled by the shell scripts. Additional details on this work, including our use of Awk and shell scripts, and information for downloading the software, can be found at www.nics.tennessee.edu/sites/default/files/HPL-site/home.html.

1. **File and function name**

   If a file or function has been modified, then its name will be changed subsequently by following the naming rules in ScaLAPACK. For example, the random number generator HPL_rand is renamed to HPL_zrand, the main header file hpl.h is changed to zhpl.h, and the routine HPL_pdgesv is renamed to HPL_pzgesv.

2. **Variable data-type**

   Nearly all the type declaration of the variables to double are changed to double complex. For example, the variable declaration of src/pgesv/HPL_rollN.c

```
void HPL_rollN (
    HPL_T_panel *        PBCST,
    double *             U,
    int *                IFLAG,
    HPL_T_panel *        PANEL,
    ...
)
```

   is changed to

```
void HPL_zrollN (
```

```
    HPL_ZT_panel *       PBCST,
    double complex *     U,
    int *                IFLAG,
    HPL_ZT_panel *       PANEL,
    ...
)
```

The datatype handles used in MPI communication are also changed correspondingly so that `MPI_Send(U, ···, MPI_DOUBLE, ···)` is changed to `MPI_Send(U, ···, MPI_DOUBLE_COMPLEX, ··· )`.

3. **Function return datatype**

   The datatype returned by a function is changed if it is determined to return a double-complex number. In `testing/matgen/HPL_rand.c`:

   ```
   double HPL_rand ( void )
   ```

   is changed to

   ```
   double complex HPL_zrand ( void )
   ```

   in `testing/matgen/HPL_zrand.c`. Others, such as those returning a norm, a time value, a coordinate, or processor number, remain unchanged.

4. **Miscellaneous**

   (a) **Operations on Complex Number**

   The original absolute value function `Mabs` is replaced by the complex norm function, `cabs`. The code in *include/hpl_misc.h*:

   ```
   #define Mabs(a_) ( ((a_)< 0) ?  -(a_):(a_) )
   ```

   is modified to

   ```
   #define Mabs(a_) cabs( (double complex) a_)
   ```

   in `include/hpl_zmisc.h`

   A pair of complex numbers cannot be compared using '>', '<', `min` or `max` functions. Their norms are compared instead. For example, in the file `src/grid/HPL_zmin.c`

   ```
    b[i] = Mmin(a[i], b[i]);
   ```

   is modified to

   ```
   if ( Mabs(a[i]) < Mabs(b[i]) ) b[i]=a[i];
   ```

where Mabs is defined to be the complex norm. This is not the most general code replacement but it works in this case.

(b) **Multi-datatype**

HPL declares a few integer values to be of type double so that they can be packed into message buffers with type double data, thus reducing message traffic. (Examples include matrix row indices and logical process grid coordinates.) In this conversion, then, those values must be declared to be type double complex. For instance, the local work space WORK and Wwork in src/pfact/HPL_pzmxswp.c are used to hold multiple data types. We need to hand-edit the prototype in communications. Here is the explanation of the variable WORK:

```
* WORK    (local workspace)             double complex *
*         On entry, WORK  is a workarray of size at least 2 * (4+2*N0).
*         WORK[0] contains  the  local  maximum  absolute value scalar,
*         WORK[1] contains  the corresponding local row index,  WORK[2]
*         contains the corresponding global row index, and  WORK[3]  is
*         the coordinate of process owning this max.  The N0 length max
*         row is stored in WORK[4:4+N0-1];  Note  that this is also the
*         JJth row  (or column) of L1. The remaining part of this array
*         is used as workspace.
```

(c) **CBLAS support**

Function of Complex BLAS (CBLAS) are called instead of the double-precision BLAS. For example, F77DSWAP is modified to be F77ZSWAP and dger_ is modified to be zgeru_. Notice that we need to distinguish betwen transpose and conjugate transpose operations.

(d) **Random number generator**

A new parallel random matrix generation algorithm is needed, as we need two double-precision numbers to construct one double-complex number. Here we use the same algorithm as ScaLAPACK PZMATGEN and double the variables jump2, jump3, jump7 in testing/pmatgen/HPL_pdmatgen.c.

(e) **Performance calculation (GFLOPS)**

The multiplication of two complex numbers requires four real multiplies and two adds and

the sum of two complex numbers requires two real adds. For the large problem sizes ($N \geq 10000$) considered, the performance calculation of GFLOPS in HPL is simply estimated to be $(4 \times 10^{-9}/3)N^3/t$, where N is the size of the matrix, t is the computation time in seconds. A more precise formula is available in the appendix of [4].

(f) **Data-Type Choice**

We prepare HPL to work on more data-types, not only `double complex`, but also `float` and `float complex` by providing extra choices on the variable `DTYPE`. For example, in `src/grid/HPL_zmin.c`

⋮

```
if( DTYPE == HPL_DOUBLE_COMPLEX )
{
    const double complex *a = (const double complex *)(IN);
    double complex *b = (double complex *)(INOUT);
    for(i = 0; i < N; i++ ) if (Mabs(a[i])<Mabs(b[i])) b[i]=a[i];
}
else if( DTYPE == HPL_COMPLEX )
{
    const float complex *a = (const float complex *)(IN);
    float complex *b = (float complex *)(INOUT);
    for(i = 0; i < N; i++ ) if (Mabs(a[i])<Mabs(b[i])) b[i]=a[i];
}
else if( DTYPE == HPL_FLOAT )
{
```

⋮

5. **Unchanged**

(a) **Timing related variables/routines**

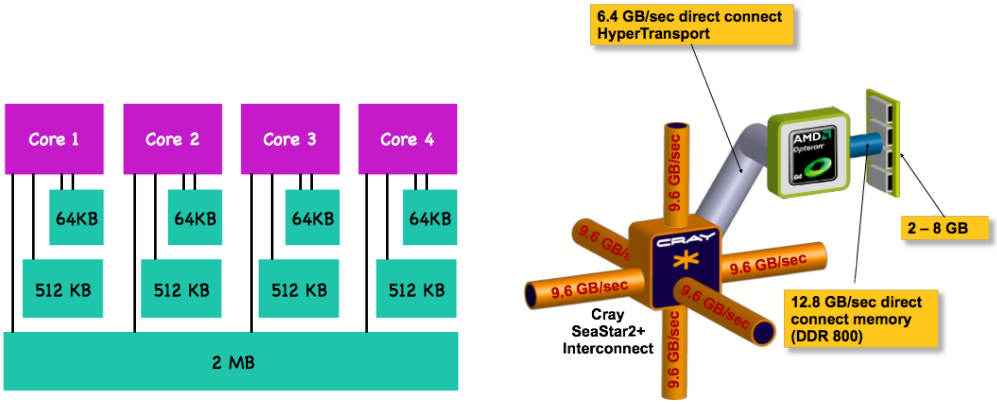
Timing routines and variables in the folders `testing/ptimer/` and `testing/timer/` are unchanged. They evaluate the computation time, and use double-precision numbers independent of datatype.

(b) **Norm related variables**

For example `Anorm1` in `HPL_pztest.c`.

(c) **Residue variables**

For example `resid0` in `testing/ptest/HPL_pztest.c`.



(a) AMD quad-core Opteron (Budapest) processor

(b) XT4 network

Fig. 5: XT4 architecture

5 Experimental platforms

Experiments were performed on two widely differing scales of compute power. In this section we describe those computing architectures.

The small platform is a Linux cluster named Organon, consisting of 9 Sun V20z servers, with one serving as the primary file server. Running the Linux operating system, the Dual AMD Opteron 2.6 GHz processors are connected by a Cisco Giga-bit switch[5].

The large platform is a Cray XT4, named Jaguar, located at the NCCS at ORNL. Jaguar is one of the largest computers in the U.S. Department of Energy's (DOE's) Office of Science and is the major computing resource for DOE's Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program[6]. It consists of 7,832 AMD Opteron 2.1 GHz quad-core processors (illustrated in Figure 5(a)) connected using SeaStar2 through HyperTransport (illustrated in Figure 5(b)) in a 3-dimensional torus topology. The operating system is a customized version of Linux, named Compute-Node Linux (CNL)[7]. This configuration provides 262 TFLOPS with 60 TBytes of memory. Early experiences with this configuration are reported in [2]. Details of both computers are summarized in Table 1.

[5] www.cuhk.edu.hk/itsc/compenv/research-computing/organon

[6] www.er.doe.gov/ascr/incite/index.html

[7] Soon to be re-named Cray Linux Environment (CLE).

Table 1: Summary of Computing Platforms

	Organon	*Jaguar*
CPU	Dual AMD Opteron 252, 2.6 GHz 1MB L2, 1GHz HyperTransport	Quad-core AMD Opteron (Budapest), 2.1 GHz 2 MB L2/core, SeaStar2 through HyperTransport
Interconnect	Cisco 3750 Switch	SeaStar 2
OS	Sun V20z	CNL
MPI	MPICH, ver. 1.2.7	Vendor library
BLAS	ATLAS, ver. 3.6.0	Cray libsci (Goto)
C Compiler	`gcc`, ver. 3.2.3	pgcc (PGI)
C flags	`-fomit-frame-pointer` `-O3 -funroll-loops`	`-fomit-frame-pointer` `-O3 -funroll-loops`
Fortran Compiler	`mpif77`	`mpif77 (PGI)`
F flags	`-O3`	`-O3`

Table 2: Performance (in GFLOPS) of HPL (`zhpl`) and ScaLAPACK (`xzlu`).

N	NB	P × Q	`zhpl`	`xzlu`	% Change
Organon cluster					
10000	72	3 × 2	20.9	17.7	17
10000	80	3 × 2	21.0	17.4	20
10000	100	3 × 2	21.1	17.8	18
Jaguar					
10000	72	3 × 2	22.4	20.1	11
10000	80	3 × 2	22.6	19.9	14
10000	100	3 × 2	22.7	20.2	12

Table 3: *Example problem sizes and associated linear system dimensions*

	matrix dim		
grid size	original	new	% reduced
350 × 350	367,500	254,823	31
400 × 400	480,000	421,206	30
450 × 450	607,500	634,593	30
500 × 500	750,000	524,475	32

6 Performance results

Two sets of experiments were constructed that compare the HPL approach with ScaLAPACK subroutine `PZGESV`. The first used the standard HPL benchmark system, set to size 10,000, permitting comparison on both platforms. The results, obtained by taking the average of two identical tests, are summarized in Table 2 . For this small test case, the HPL solver achieved better performance by 11% to 20% over ScaLAPACK. On both machines, best results were obtained with the larger block sizes (`NB`), attributable to the large cache. However, HPL shows stronger performance relative to ScaLAPACK for the middle block size, also on both machines. Jaguar outperformed Organon in spite of its slower clock speed, which we attribute to Jaguar's faster interconnect.

The second set was configured within the context of large scale AORSA simulations, and thus can only fit on a leadership class computer. Problems operated on grids of size 350×350, 400×400, 450×450, and 500×500. The latter is the largest resolution thus far simulated by AORSA. The dimensions of the linear systems generated are shown in Table 3 including the dimensions that would have been generated without the equation reduction strategy described in Section 3.1. Results are summarized in Table 4.

Two MPI processes were mapped to each node, one MPI process per Opteron core. As previously

Table 4: Performance of ScaLAPACK and HPL for a set of ITER simulations.

modes	description	performance (TFlops/sec)
500×500	HPL + Goto BLAS	87.5
400×400	HPL	69.2
350×350	HPL	64.6
450×450	ScaLAPACK	47.2
400×400	ScaLAPACK	42.6
350×350	ScaLAPACK	38.0

shown in Figure 1, the ScaLAPACK solver achieved good performance and scalability up to nearly 5,000 MPI processes but performance levels off on 10,000 and 20,000 processes. Using the HPL solver, performance increased from 38 TFLOPS to 69.2 TFLOPS. The highest performance of 87.5 TFLOPS was achieved on solving a large problem with 500×500 modes and replacing the vendor BLAS with the Goto optimized BLAS [19]. As stated above, access to Jaguar is controlled through awarded allocations, which limits our ability to test different configurations, and thus one-to-one comparisons are not always possible. However, the trends are clear, and in fact led Cray to incorporate the Goto BLAS into their scientific library.

To gain a better insight into why HPL is more efficient than ScaLAPACK, we ran the default HPL driver to exercise several algorithmic options. We experimented on Jaguar to solve double precision matrices of size 50,000 using 256 cores on a square 16×16 processor grid. The achieved performances ranged from 1.13 TFLOPS to 1.26 TFLOPS. The largest improvement of nearly 10% was produced by using look ahead versus no look ahead (note ScaLAPACK has no look ahead algorithm). Further small improvements were gained by tuning the methods for communication and broadcast. We postulate that the look ahead algorithm for shortening the critical path is even more effective when larger processor configurations are used.

7 Summary

We have modified the High Performance LINPACK benchmark, known as HPL, to accept double precision (64-bit) complex coefficients, and have created an interface for its use by AORSA, a computer program used to simulate current flow, heating and control of instabilities in a plasma in a fusion energy device. The result is a significant increase in the performance of the dominant runtime computational kernel in AORSA, the solution of the dense linear system.

These changes have resulted in an interface analogous to that provided by ScaLAPACK. In particular, the matrix is assumed to be distributed in 2D block-cyclic format, contained in ScaLAPACK defined variables and data structures. We hope this will encourage vendors to include this capability in their libraries. Further, by documenting the steps involved in this conversion, we believe that the same approach can be used to generate a float or single-precision complex version of the HPL solver.

As computing capabilities increase, enabling even greater resolution of the simulations, we continue to investigate algorithmic as well as computational improvements to this and other components of AORSA. Future efforts may incorporate the use of mixed precision iterative solvers, where the LU factorization is computed with reduced precision in HPL and used as a preconditioner in an iterative method. For matrices that are not too ill-conditioned, this approach has been shown to produce significant performance benefits [9, 2].

References

[1] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.

[2] S.R. Alam, R.F. Barrett, M. Eisenbach, M.R. Fahey, R. Hartman-Baker, J.A. Kuehn, S.W. Poole, R. Sankaran, and P.H. Worley. The Cray XT4 Quad-core : A First Look. In *Proc. 50th Cray User Group meeting*, Helsinki, FI, May 2008.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. Also available at www.netlib.org/lapack/lug/.

[4] E. Anderson and J.J. Dongarra. LAPACK working note 18: Implementation guide for LAPACK. Technical Report UT-CS-90-101, University of Tennessee, Knoxville, Tennessee, 1990. Also available at www.netlib.org/lapack/lawns/downloads/.

[5] R. Aymar, V. A. Chuyanov, M. Huguet, and Y. Shimomura. Overview of ITER-FEAT - the future international burning plasma experiment. *Nuclear Fusion*, 41(10), 2001.

[6] R.F. Barrett, T. Chan, E.F. D'Azevedo, E.F. Jaeger, K. Wong, and R. Wong. A complex-variables version of high performance computing LINPACK benchmark (HPL). *Concurrency and Computation: Practice and Exerience*, 2009. To appear.

[7] R.H. Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, USA, 2004.

[8] L.S. Blackford, J. Choi, A. Cleary, E.F. D'Azevedo, J. Demmel, I. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997. Also available at www.netlib.org/scalapack/slug/index.html.

[9] A. Buttari, J.J. Dongarra, J. Langou, P. Luszczek, and J. Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications*, 21(4):457–466, 2007.

[10] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers – design issues and performance. *Computer Physics Communications*, 97(1-2):1–15, 1996.

[11] J. Choi, J.J. Dongarra, A. Petitet, D. Walker, and R.C. Whaley. A Proposal for a Set of Parallel, Basic Linear Algebra Subprograms (PBLAS), LAPACK working note 100. Technical Report CS-94-239, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1994. Also available at www.netlib.org/lapack/lawns/downloads/.

[12] J. Demmel, J.J. Dongarra, B. Parlett, W. Kahan, M. Gu, D. Bindel, Y. Hida, X. Li, O. Marques, E.J. Riedy, C. Voemel, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, J. Langou, and S. Tomov. Prospectus for the Next LAPACK and ScaLAPACK Libraries. In *PARA 2006*, Uma, Sweden, June 2006.

[13] J.J. Dongarra. Performance of various computers using standard linear equation software. Technical Report CS-89-85, Computer Science Department; University of Tennessee, Knoxville, Tennessee, 1989.

[14] J.J. Dongarra, J. Du Croz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 16:1–17, 1990.

[15] J.J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Trans.on Math. Soft.*, 14:1–32, 1988.

[16] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998.

[17] J.J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurrency Computat.: Pract. Exper.*, 15:803–820, 2003. The HPL software is available at www.netlib.org/benchmark/hpl.

[18] J.J. Dongarra and R.C. Whaley. LAPACK working note 94: A users' guide to the BLACS. Technical Report UT-CS-95-281, Computer Science Department, University of Tennessee, Knoxville, Tennessee, 1997.

[19] K. Goto and R.A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(4), May 2008.

[20] E.F. Jaeger, L.A. Berry, E.F. D'Azevedo, D.B. Batchelor, M.D. Carter, K.F. White, and H. Weitzner. Advances in full-wave modeling of radio frequency heated multidimensional plasmas. *Physics of Plasmas*, 9(5):1873–1881, 2002.

[21] E.F. Jaeger, L.A. Berry, J.R. Myra, D.B. Batchelor, E.F. D'Azevedo, P.T. Bonoli, C.K. Philips, D.N. Smithe, D.A. D'Ippolito, M.D. Carter, R.J. Dumont, J.C. Wright, and R.W. Harvey. Sheared poloidal flow driven by mode conversion in tokamak plasmas. *Phys. Rev. Lett.*, 90(19), 2003.

[22] E.F. Jaeger, R. W. Harvey, L. A. Berry, J. R. Myra, R. J. Dumont, C. K. Philips, D. N. Smithe, R. F. Barrett, D. B. Batchelor, P. T. Bonoli, M. D. Carter, E.F. D'Azevedo, D. A. D'Ippolito, R. D. Moore, and J. C. Wright. Global-wave solutions with self-consistent velocity distributions in ion cyclotron heated plasmas. *Nuclear Fusion*, 46(7):S397–S408, 2006.

[23] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans.on Math. Soft.*, 5:308–325, 1979.

[24] M. Lutz. *Programming Python*. O'Reilly Media, third edition edition, 2006.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J.J. Dongarra. *MPI: The Complete Reference: Volume 1 - 2nd Edition*. The MIT Press, 1998.

[26] P. E. Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorizations. *Int. J. Parallel Distrib. Systems Networks*, 4(1):26–35, 2001.

[27] R.A. van de Geijn. *Using PLAPACK*. MIT Press, Cambridge, MA, 1997.

[28] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, third edition edition, 2000.

[29] R.C. Whaley and J.J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998. www.cs.utsa.edu/~whaley/papers/atlas_sc98.ps.

Table 5: Additional Files list

File name	Usage	Remarks
`Make.test`	Makefile for compilation of double-complex HPL and makes the executable file `zhpl`.	Need User Input
`Patch.sh`	The Master script which will call files in the folder '`patch/script/`' to apply changes to existing HPL files.	Need User Input
`Master.awk`	The file contains the information for changing variable/function names.	
`script/*.sh`	Applying '`master.awk`' to each file and does specific changes in the exceptional cases.	

Appendix 1: Compilation steps

The awk and shell scripts (`patch.tar`) is available by contacting the author at `e6d@ornl.gov`. The compilation of the executable file `zhpl` has several steps to modify the original HPL code and re-compile the new complex version. The detailed procedure are as follows:

1. Install the original HPL,
2. untar the patch file '`patch.tar`' to `hpl/patch`, or elsewhere,
3. input correct paths on the header of `patch.sh`,
4. run '`sh patch.sh`', which will unpack the extension of files for you,
5. input correct path names on header of `Make.test`, so Makefile will run smoothly,
6. run '`make -f Make.test arch=UNKNOWN`' to build the executable code "zhpl".

Table 5 describes the additional files associated with the complex HPL.

Appendix 2: Awk script

```
#########################################################################
### Master.awk                                                        ###
### This file contains the information for changing the function      ###
### and variable names.                                               ###
BEGIN {       nlines = 0;


# Global Dictionary

   # widely used variables
         name_dictionary["MPI_DOUBLE"] = "MPI_DOUBLE_COMPLEX";
         name_dictionary["HPL_DOUBLE"] = "HPL_DOUBLE_COMPLEX";



   # panel variables
         name_dictionary["HPL_T_panel"] = "HPL_ZT_panel";
         name_dictionary["HPL_T_palg"] = "HPL_ZT_palg";
         name_dictionary["HPL_S_palg"] = "HPL_ZS_palg";
         name_dictionary["HPL_T_pmat"] = "HPL_ZT_pmat";
         name_dictionary["HPL_S_pmat"] = "HPL_ZS_pmat";

   # pfact variables
         name_dictionary["HPL_T_UPD_FUN"] = "HPL_ZT_UPD_FUN";
         name_dictionary["HPL_T_RFA_FUN"] = "HPL_ZT_RFA_FUN";
         name_dictionary["HPL_T_PFA_FUN"] = "HPL_ZT_PFA_FUN";
         name_dictionary["HPL_S_PANEL"] = "HPL_ZS_PANEL";
         name_dictionary["HPL_T_TYPE"] = "HPL_ZT_TYPE";

   #testing/ptest
         name_dictionary["HPL_pddriver"] = "HPL_pzdriver";
         name_dictionary["HPL_pdtest"] = "HPL_pztest";
         name_dictionary["HPL_pdinfo"] = "HPL_pzinfo";

   #testing/pmatgen
         name_dictionary["HPL_pdmatgen"] = "HPL_pzmatgen";
```

```
#testing/matgen
      name_dictionary["HPL_dmatgen"] = "HPL_zmatgen";
      name_dictionary["HPL_rand"] = "HPL_zrand";

#include/
      name_dictionary["hpl_auxil.h"] = "hpl_zauxil.h";
      name_dictionary["hpl_blas.h"] = "hpl_zblas.h";
      name_dictionary["hpl_comm.h"] = "hpl_zcomm.h";
      name_dictionary["hpl_gesv.h"] = "hpl_zgesv.h";
      name_dictionary["hpl_grid.h"] = "hpl_zgrid.h";
      name_dictionary["hpl.h"] = "zhpl.h";
      name_dictionary["hpl_matgen.h"] = "hpl_zmatgen.h";
      name_dictionary["hpl_misc.h"] = "hpl_zmisc.h";
      name_dictionary["hpl_panel.h"] = "hpl_zpanel.h";
      name_dictionary["hpl_pauxil.h"] = "hpl_pzauxil.h";
      name_dictionary["hpl_pfact.h"] = "hpl_pzfact.h";
      name_dictionary["hpl_pgesv.h"] = "hpl_pzgesv.h";
      name_dictionary["hpl_pmatgen.h"] = "hpl_pzmatgen.h";
      name_dictionary["hpl_pmisc.h"] = "hpl_pzmisc.h";
      name_dictionary["hpl_ptest.h"] = "hpl_pztest.h";
      name_dictionary["hpl_test.h"] = "hpl_ztest.h";
      name_dictionary["hpl_units.h"] = "hpl_zunits.h";

# src/auxil
      name_dictionary["HPL_dlacpy"] = "HPL_zlacpy";
      name_dictionary["HPL_dlange"] = "HPL_zlange";
      name_dictionary["HPL_dlaprnt"] = "HPL_zlaprnt";
      name_dictionary["HPL_dlatcpy"] = "HPL_zlatcpy";

# src/blas
      name_dictionary["HPL_dgemmNN"] = "HPL_zgemmNN";
      name_dictionary["HPL_dgemmNT"] = "HPL_zgemmNT";
      name_dictionary["HPL_dgemmTN"] = "HPL_zgemmTN";
      name_dictionary["HPL_dgemmTT"] = "HPL_zgemmTT";
      name_dictionary["HPL_dgemm0"] = "HPL_zgemm0";
      name_dictionary["HPL_dtrsmLLNN"] = "HPL_ztrsmLLNN";
```

```
name_dictionary["HPL_dtrsmLLNU"] = "HPL_ztrsmLLNU";
name_dictionary["HPL_dtrsmLLTN"] = "HPL_ztrsmLLTN";
name_dictionary["HPL_dtrsmLLTU"] = "HPL_ztrsmLLTU";
name_dictionary["HPL_dtrsmLUNN"] = "HPL_ztrsmLUNN";
name_dictionary["HPL_dtrsmLUNU"] = "HPL_ztrsmLUNU";
name_dictionary["HPL_dtrsmLUTN"] = "HPL_ztrsmLUTN";
name_dictionary["HPL_dtrsmLUTU"] = "HPL_ztrsmLUTU";
name_dictionary["HPL_dtrsmRLNN"] = "HPL_ztrsmRLNN";
name_dictionary["HPL_dtrsmRLNU"] = "HPL_ztrsmRLNU";
name_dictionary["HPL_dtrsmRLTN"] = "HPL_ztrsmRLTN";
name_dictionary["HPL_dtrsmRLTU"] = "HPL_ztrsmRLTU";
name_dictionary["HPL_dtrsmRUNN"] = "HPL_ztrsmRUNN";
name_dictionary["HPL_dtrsmRUNU"] = "HPL_ztrsmRUNU";
name_dictionary["HPL_dtrsmRUTN"] = "HPL_ztrsmRUTN";
name_dictionary["HPL_dtrsmRUTU"] = "HPL_ztrsmRUTU";
name_dictionary["HPL_dtrsm0"] = "HPL_ztrsm0";
name_dictionary["HPL_dtrsvLNN"] = "HPL_ztrsvLNN";
name_dictionary["HPL_dtrsvLNU"] = "HPL_ztrsvLNU";
name_dictionary["HPL_dtrsvLTN"] = "HPL_ztrsvLTN";
name_dictionary["HPL_dtrsvLTU"] = "HPL_ztrsvLTU";
name_dictionary["HPL_dtrsvUNN"] = "HPL_ztrsvUNN";
name_dictionary["HPL_dtrsvUNU"] = "HPL_ztrsvUNU";
name_dictionary["HPL_dtrsvUTN"] = "HPL_ztrsvUTN";
name_dictionary["HPL_dtrsvUTU"] = "HPL_ztrsvUTU";
name_dictionary["HPL_dtrsv0"] = "HPL_ztrsv0";
name_dictionary["HPL_dgemv0"] = "HPL_zgemv0";

name_dictionary["HPL_daxpy"] = "HPL_zaxpy";
name_dictionary["HPL_dcopy"] = "HPL_zcopy";
name_dictionary["HPL_dgemm"] = "HPL_zgemm";
name_dictionary["HPL_dgemv"] = "HPL_zgemv";
name_dictionary["HPL_dger"] = "HPL_zger";
name_dictionary["HPL_dscal"] = "HPL_zscal";
name_dictionary["HPL_dswap"] = "HPL_zswap";
name_dictionary["HPL_dtrsm"] = "HPL_ztrsm";
name_dictionary["HPL_dtrsv"] = "HPL_ztrsv";
name_dictionary["HPL_idamax"] = "HPL_izamax";
```

```
#########################################################################
### NOTICE ...
### To further extend the precisions of zHPL, need to add different
### type of F77 blacs functions to it.
#########################################################################

      name_dictionary["F77daxpy"] = "F77zaxpy";
      name_dictionary["F77dcopy"] = "F77zcopy";
      name_dictionary["F77dgemm"] = "F77zgemm";
      name_dictionary["F77dgemv"] = "F77zgemv";
      name_dictionary["F77dger"] = "F77zger";
      name_dictionary["F77dscal"] = "F77zscal";
      name_dictionary["F77dswap"] = "F77zswap";
      name_dictionary["F77dtrsm"] = "F77ztrsm";
      name_dictionary["F77dtrsv"] = "F77ztrsv";
      name_dictionary["F77idamax"] = "F77izamax";

#########################################################################

# src/comm
      name_dictionary["HPL_1ring"] = "HPL_z1ring";
      name_dictionary["HPL_1rinM"] = "HPL_z1rinM";
      name_dictionary["HPL_2ring"] = "HPL_z2ring";
      name_dictionary["HPL_2rinM"] = "HPL_z2rinM";
      name_dictionary["HPL_bcast"] = "HPL_zbcast";
      name_dictionary["HPL_binit"] = "HPL_zbinit";
      name_dictionary["HPL_blong"] = "HPL_zblong";
      name_dictionary["HPL_blonM"] = "HPL_zblonM";
      name_dictionary["HPL_bwait"] = "HPL_zbwait";
      name_dictionary["HPL_copyL"] = "HPL_zcopyL";
      name_dictionary["HPL_packL"] = "HPL_zpackL";
      name_dictionary["HPL_recv"] = "HPL_zrecv";
      name_dictionary["HPL_sdrv"] = "HPL_zsdrv";
      name_dictionary["HPL_send"] = "HPL_zsend";

# subfunctions within /src/grid/*.c
```

```
      name_dictionary["HPL_binit_blong"] = "HPL_zbinit_blong";
      name_dictionary["HPL_bcast_blong"] = "HPL_zbcast_blong";
      name_dictionary["HPL_bwait_blong"] = "HPL_zbwait_blong";
      name_dictionary["HPL_binit_blonM"] = "HPL_zbinit_blonM";
      name_dictionary["HPL_bcast_blonM"] = "HPL_zbcast_blonM";
      name_dictionary["HPL_bwait_blonM"] = "HPL_zbwait_blonM";
      name_dictionary["HPL_binit_1ring"] = "HPL_zbinit_1ring";
      name_dictionary["HPL_bcast_1ring"] = "HPL_zbcast_1ring";
      name_dictionary["HPL_bwait_1ring"] = "HPL_zbwait_1ring";
      name_dictionary["HPL_binit_1rinM"] = "HPL_zbinit_1rinM";
      name_dictionary["HPL_bcast_1rinM"] = "HPL_zbcast_1rinM";
      name_dictionary["HPL_bwait_1rinM"] = "HPL_zbwait_1rinM";
      name_dictionary["HPL_binit_2ring"] = "HPL_zbinit_2ring";
      name_dictionary["HPL_bcast_2ring"] = "HPL_zbcast_2ring";
      name_dictionary["HPL_bwait_2ring"] = "HPL_zbwait_2ring";
      name_dictionary["HPL_binit_2rinM"] = "HPL_zbinit_2rinM";
      name_dictionary["HPL_bcast_2rinM"] = "HPL_zbcast_2rinM";
      name_dictionary["HPL_bwait_2rinM"] = "HPL_zbwait_2rinM";

# src/grid
      name_dictionary["HPL_all_reduce"] = "HPL_zall_reduce";
      name_dictionary["HPL_broadcast"] = "HPL_zbroadcast";
      name_dictionary["HPL_max"] = "HPL_zmax";
      name_dictionary["HPL_min"] = "HPL_zmin";
      name_dictionary["HPL_reduce"] = "HPL_zreduce";
      name_dictionary["HPL_sum"] = "HPL_zsum";

# src/panel
      name_dictionary["HPL_pdpanel_disp"] = "HPL_pzpanel_disp";
      name_dictionary["HPL_pdpanel_free"] = "HPL_pzpanel_free";
      name_dictionary["HPL_pdpanel_init"] = "HPL_pzpanel_init";
      name_dictionary["HPL_pdpanel_new"] = "HPL_pzpanel_new";

# src/pauxil
      name_dictionary["HPL_dlaswp00N"] = "HPL_zlaswp00N";
      name_dictionary["HPL_dlaswp01N"] = "HPL_zlaswp01N";
      name_dictionary["HPL_dlaswp01T"] = "HPL_zlaswp01T";
```

```
      name_dictionary["HPL_dlaswp02N"] = "HPL_zlaswp02N";
      name_dictionary["HPL_dlaswp03N"] = "HPL_zlaswp03N";
      name_dictionary["HPL_dlaswp03T"] = "HPL_zlaswp03T";
      name_dictionary["HPL_dlaswp04N"] = "HPL_zlaswp04N";
      name_dictionary["HPL_dlaswp04T"] = "HPL_zlaswp04T";
      name_dictionary["HPL_dlaswp05N"] = "HPL_zlaswp05N";
      name_dictionary["HPL_dlaswp05T"] = "HPL_zlaswp05T";
      name_dictionary["HPL_dlaswp06N"] = "HPL_zlaswp06N";
      name_dictionary["HPL_dlaswp06T"] = "HPL_zlaswp06T";
      name_dictionary["HPL_dlaswp10N"] = "HPL_zlaswp10N";
      name_dictionary["HPL_pdlange"] = "HPL_pzlange";
      name_dictionary["HPL_pdlaprnt"] = "HPL_pzlaprnt";

# src/pfact
      name_dictionary["HPL_dlocmax"] = "HPL_zlocmax";
      name_dictionary["HPL_dlocswpN"] = "HPL_zlocswpN";
      name_dictionary["HPL_dlocswpT"] = "HPL_zlocswpT";
      name_dictionary["HPL_pdfact"] = "HPL_pzfact";
      name_dictionary["HPL_pdmxswp"] = "HPL_pzmxswp";
      name_dictionary["HPL_pdpancrN"] = "HPL_pzpancrN";
      name_dictionary["HPL_pdpancrT"] = "HPL_pzpancrT";
      name_dictionary["HPL_pdpanllN"] = "HPL_pzpanllN";
      name_dictionary["HPL_pdpanllT"] = "HPL_pzpanllT";
      name_dictionary["HPL_pdpanrlN"] = "HPL_pzpanrlN";
      name_dictionary["HPL_pdpanrlT"] = "HPL_pzpanrlT";
      name_dictionary["HPL_pdrpancrN"] = "HPL_pzrpancrN";
      name_dictionary["HPL_pdrpancrT"] = "HPL_pzrpancrT";
      name_dictionary["HPL_pdrpanllN"] = "HPL_pzrpanllN";
      name_dictionary["HPL_pdrpanllT"] = "HPL_pzrpanllT";
      name_dictionary["HPL_pdrpanrlN"] = "HPL_pzrpanrlN";
      name_dictionary["HPL_pdrpanrlT"] = "HPL_pzrpanrlT";

# src/pgesv
      name_dictionary["HPL_dgesv"] = "HPL_zgesv";
      name_dictionary["HPL_pdgesv0"] = "HPL_pzgesv0";
      name_dictionary["HPL_pdgesv"] = "HPL_pzgesv";
      name_dictionary["HPL_pdgesvK1"] = "HPL_pzgesvK1";
```

```
        name_dictionary["HPL_pdgesvK2"] = "HPL_pzgesvK2";
        name_dictionary["HPL_pdlaswp00N"] = "HPL_pzlaswp00N";
        name_dictionary["HPL_pdlaswp00T"] = "HPL_pzlaswp00T";
        name_dictionary["HPL_pdlaswp01N"] = "HPL_pzlaswp01N";
        name_dictionary["HPL_pdlaswp01T"] = "HPL_pzlaswp01T";
        name_dictionary["HPL_pdlaswp01"] = "HPL_pzlaswp01";
        name_dictionary["HPL_pdlaswp00"] = "HPL_pzlaswp00";
        name_dictionary["HPL_pdtrsv"] = "HPL_pztrsv";
        name_dictionary["HPL_pdupdateNN"] = "HPL_pzupdateNN";
        name_dictionary["HPL_pdupdateNT"] = "HPL_pzupdateNT";
        name_dictionary["HPL_pdupdateTN"] = "HPL_pzupdateTN";
        name_dictionary["HPL_pdupdateTT"] = "HPL_pzupdateTT";
        name_dictionary["HPL_pdupdate"] = "HPL_pzupdate";
        name_dictionary["HPL_pipid"] = "HPL_pzipid";
        name_dictionary["HPL_plindx0"] = "HPL_pzlindx0";
        name_dictionary["HPL_plindx10"] = "HPL_pzlindx10";
        name_dictionary["HPL_plindx1"] = "HPL_pzlindx1";
        name_dictionary["HPL_rollN"] = "HPL_zrollN";
        name_dictionary["HPL_rollT"] = "HPL_zrollT";
        name_dictionary["HPL_spreadN"] = "HPL_zspreadN";
        name_dictionary["HPL_spreadT"] = "HPL_zspreadT";
        name_dictionary["HPL_equil"] = "HPL_zequil";

    # miscellaneous
        name_dictionary["xhpl"] = "zhpl";

        name_dictionary["HPL_pdelset"] = "HPL_pzelset";
        name_dictionary["HPL_pdelget"] = "HPL_pzelget";

}


# ------------------------------------
# wish to avoid changing /^double/
# ------------------------------------
/^double/ {
          $1 = "__UGLY_HACK__";
```

```
 }

/\ydouble\y/ {     #\y is word boundary
        gsub("\\ydouble\\y","zcomplex");
}

/include "hpl.h"/ {
        gsub("hpl.h","zhpl.h");
}

{
        for (i in name_dictionary) {
                pattern = "\\y" i "\\y";
                value = name_dictionary[i];

                gsub( pattern, value );
        };


        # ---------------------------------------
        # save the input for post-processing
        # ---------------------------------------

        nlines = nlines + 1;
        lines[ nlines ]  = $0;
}

END {
        for(i=1; i <= nlines; i += 1) {
                # ---------------
                # undo ugly hack
                # ---------------
                sub(/^__UGLY_HACK__/,"double", lines[i]);
                print lines[i];
        };
}
```

Appendix 3: List of modified files

Here is the list of files produced in the complex version of HPL. The original name is in brackets.

- Executable File: zhpl (xhpl)
- Library Archive: libzhpl.a (libhpl.a)
- hpl/include
    1. hpl_ptimer.h
    2. hpl_pzauxil.h (hpl_pauxil.h)
    3. hpl_pzfact.h (hpl_pfact.h)
    4. hpl_pzgesv.h (hpl_pgesv.h)
    5. hpl_pzmatgen.h (hpl_pmatgen.h)
    6. hpl_pzmisc.h (hpl_pmisc.h)
    7. hpl_pztest.h (hpl_ptest.h)
    8. hpl_timer.h
    9. hpl_zauxil.h (hpl_auxil.h)
    10. hpl_zblas.h (hpl_blas.h)
    11. hpl_zcomm.h (hpl_comm.h)
    12. hpl_zgesv.h (hpl_gesv.h)
    13. hpl_zgrid.h (hpl_grid.h)
    14. hpl_zmatgen.h (hpl_matgen.h)
    15. hpl_zmisc.h (hpl_misc.h)
    16. hpl_zpanel.h (hpl_panel.h)
    17. hpl_ztest.h (hpl_test.h)
    18. hpl_zunits.h (hpl_units.h)
    19. zhpl.h (hpl.h)
- hpl/src/auxil
    1. HPL_abort.c
    2. HPL_dlamch.c
    3. HPL_fprintf.c

4. HPL_warn.c
5. HPL_zlacpy.c (HPL_dlacpy.c)
6. HPL_zlange.c (HPL_dlange.c)
7. HPL_zlaprnt.c (HPL_dlaprnt.c)
8. HPL_zlatcpy.c (HPL_dlatcpy.c)

- hpl/src/blas

1. HPL_izamax.c (HPL_idamax.c)
2. HPL_zaxpy.c (HPL_daxpy.c)
3. HPL_zcopy.c (HPL_dcopy.c)
4. HPL_zgemm.c (HPL_dgemm.c)
5. HPL_zgemv.c (HPL_dgemv.c)
6. HPL_zger.c (HPL_dger.c)
7. HPL_zscal.c (HPL_dscal.c)
8. HPL_zswap.c (HPL_dswap.c)
9. HPL_ztrsm.c (HPL_dtrsm.c)
10. HPL_ztrsv.c (HPL_dtrsv.c)

- hpl/src/comm

1. HPL_z1ring.c (HPL_1ring.c)
2. HPL_z1rinM.c (HPL_1rinM.c)
3. HPL_z2ring.c (HPL_2ring.c)
4. HPL_z2rinM.c (HPL_2rinM.c)
5. HPL_zbcast.c (HPL_bcast.c)
6. HPL_zbinit.c (HPL_binit.c)
7. HPL_zblong.c (HPL_blong.c)
8. HPL_zblonM.c (HPL_blonM.c)
9. HPL_zbwait.c (HPL_bwait.c)
10. HPL_zcopyL.c (HPL_copyL.c)
11. HPL_zpackL.c (HPL_packL.c)
12. HPL_zrecv.c (HPL_recv.c)

13. HPL_zsdrv.c (HPL_sdrv.c)
14. HPL_zsend.c (HPL_send.c)

- hpl/src/grid

1. HPL_barrier.c
2. HPL_grid_exit.c
3. HPL_grid_info.c
4. HPL_grid_init.c
5. HPL_pnum.c
6. HPL_zall_reduce.c (HPL_all_reduce.c)
7. HPL_zbroadcast.c (HPL_broadcast.c)
8. HPL_zmax.c (HPL_max.c)
9. HPL_zmin.c(HPL_min.c)
10. HPL_zreduce.c (HPL_reduce.c)
11. HPL_zsum.c (HPL_sum.c)

- hpl/src/panel

1. HPL_pzpanel_disp.c (HPL_pdpanel_disp.c)
2. HPL_pzpanel_free.c (HPL_pdpanel_free.c)
3. HPL_pzpanel_init.c (HPL_pdpanel_init.c)
4. HPL_pzpanel_new.c (HPL_pdpanel_new.c)

- hpl/src/pauxil

1. HPL_indxg2l.c
2. HPL_indxg2lp.c
3. HPL_indxg2p.c
4. HPL_indxl2g.c
5. HPL_infog2l.c
6. HPL_numroc.c
7. HPL_numrocI.c
8. HPL_pabort.c

9. HPL_pdlamch.c
10. HPL_pwarn.c
11. HPL_pzlange.c (HPL_pdlange.c)
12. HPL_pzlaprnt.c (HPL_pdlaprnt.c)
13. HPL_zlaswp00N.c (HPL_dlaswp00N.c)
14. HPL_zlaswp01N.c (HPL_dlaswp01N.c)
15. HPL_zlaswp01T.c (HPL_dlaswp01T.c)
16. HPL_zlaswp02N.c (HPL_dlaswp02N.c)
17. HPL_zlaswp03N.c (HPL_dlaswp03N.c)
18. HPL_zlaswp03T.c (HPL_dlaswp03T.c)
19. HPL_zlaswp04N.c (HPL_dlaswp04N.c)
20. HPL_zlaswp04T.c (HPL_dlaswp04T.c)
21. HPL_zlaswp05N.c (HPL_dlaswp05N.c)
22. HPL_zlaswp05T.c (HPL_dlaswp05T.c)
23. HPL_zlaswp06N.c (HPL_dlaswp06N.c)
24. HPL_zlaswp06T.c (HPL_dlaswp06T.c)
25. HPL_zlaswp10N.c (HPL_dlaswp10N.c)

- hpl/src/pfact

1. HPL_pzfact.c (HPL_pdfact.c)
2. HPL_pzmxswp.c (HPL_pdmxswp.c)
3. HPL_pzpancrN.c (HPL_pdpancrN.c)
4. HPL_pzpancrT.c (HPL_pdpancrT.c)
5. HPL_pzpanllN.c (HPL_pdpanllN.c)
6. HPL_pzpanllT.c (HPL_pdpanllT.c)
7. HPL_pzpanrlN.c (HPL_pdpanrlN.c)
8. HPL_pzpanrlT.c (HPL_pdpanrlT.c)
9. HPL_pzrpancrN.c (HPL_pdrpancrN.c)
10. HPL_pzrpancrT.c (HPL_pdrpancrT.c)
11. HPL_pzrpanllN.c (HPL_pdrpanllN.c)
12. HPL_pzrpanllT.c (HPL_pdrpanllT.c)

13. HPL_pzrpanrlN.c (HPL_pdrpanrlN.c)
14. HPL_pzrpanrlT.c (HPL_pdrpanrlT.c)
15. HPL_zlocmax.c (HPL_dlocmax.c)
16. HPL_zlocswpN.c (HPL_dlocswpN.c)
17. HPL_zlocswpT.c (HPL_dlocswpT.c)

- hpl/src/pgesv

1. HPL_equil.c
2. HPL_logsort.c
3. HPL_perm.c
4. HPL_pzgesv.c (HPL_pdgesv.c)
5. HPL_pzgesv0.c (HPL_pdgesv0.c)
6. HPL_pzgesvK1.c (HPL_pdgesvK1.c
7. HPL_pzgesvK2.c (HPL_pdgesvK2.c)
8. HPL_pzipid.c (HPL_pipid.c)
9. HPL_pzlaswp00N.c (HPL_pdlaswp00N.c)
10. HPL_pzlaswp00T.c (HPL_pdlaswp00T.c)
11. HPL_pzlaswp01N.c (HPL_pdlaswp01N.c)
12. HPL_pzlaswp01T.c (HPL_pdlaswp01T.c)
13. HPL_pzlindx0.c (HPL_plindx0.c)
14. HPL_pzlindx1.c (HPL_plindx1.c)
15. HPL_pzlindx10.c (HPL_plindx10.c)
16. HPL_pztrsv.c (HPL_pdtrsv.c)
17. HPL_pzupdateNN.c (HPL_pdupdateNN.c)
18. HPL_pzupdateNT.c (HPL_pdupdateNT.c)
19. HPL_pzupdateTN.c (HPL_pdupdateTN.c)
20. HPL_pzupdateTT.c (HPL_pdupdateTT.c)
21. HPL_zequil.c (HPL_equil.c)
22. HPL_zrollN.c (HPL_rollN.c)
23. HPL_zrollT.c (HPL_rollT.c)
24. HPL_zspreadN.c (HPL_spreadN.c)

25. HPL_zspreadT.c (HPL_spreadT.c)

- hpl/testing/matgen

1. HPL_jumpit.c
2. HPL_ladd.c
3. HPL_lmul.c
4. HPL_setran.c
5. HPL_xjumpm.c
6. HPL_zmatgen.c (HPL_matgen.c)
7. HPL_zrand.c (HPL_rand.c)

- hpl/testing/pmatgen

1. HPL_pzmatgen.c (HPL_pdmatgen.c)

- hpl/testing/ptest

1. HPL_pzdriver.c (HPL_pddriver.c)
2. HPL_pzinfo.c (HPL_pdinfo.c)
3. HPL_pztest.c (HPL_pdtest.c)

- hpl/testing/ptimer

1. HPL_ptimer.c
2. HPL_ptimer_cputime.c
3. HPL_ptimer_walltime.c

- hpl/testing/timer

1. HPL_timer.c
2. HPL_timer_cputime.c
3. HPL_timer_walltime.c